Working with filesystem in Time Series databases

Dmytro Kozlov
Software Engineer

About me

- **Dmytro Kozlov**
  Software Engineer at VictoriaMetrics
- Working in VictoriaMetrics cloud team

Why this matters

- Average install: **2–4M samples/s** ingestion
- Large setups: **100M+ samples/s** on a single cluster
- VictoriaLogs: same storage ideas applied to logs
- Written **from scratch in Go**, open source

Agenda

- Why disk matters for a TSDB
- Write path: page cache, fsync, parts, WAL
- Write amplification: merges and partitions
- Surviving power outages
- NFS
- Instant snapshots
- Read path & the new feature: FADV_RANDOM
- Summary

Disk is slow

Why a TSDB is all about the disk

The bottleneck

- CPU
- RAM
- **Disk**
- Concurrency
- Indexing

Most bottlenecks come from **disk being slooow…**

Operation	Latency
RAM access	~100 ns
NVMe read	~20–100 µs
fsync on SSD	0.1–4 ms
fsync on HDD	~20 ms

TSDB write workload

- **Many series**, small samples, constant stream
- Mostly writes for **"now"**
- **Append-only.** No updates in place

TSDB read workload

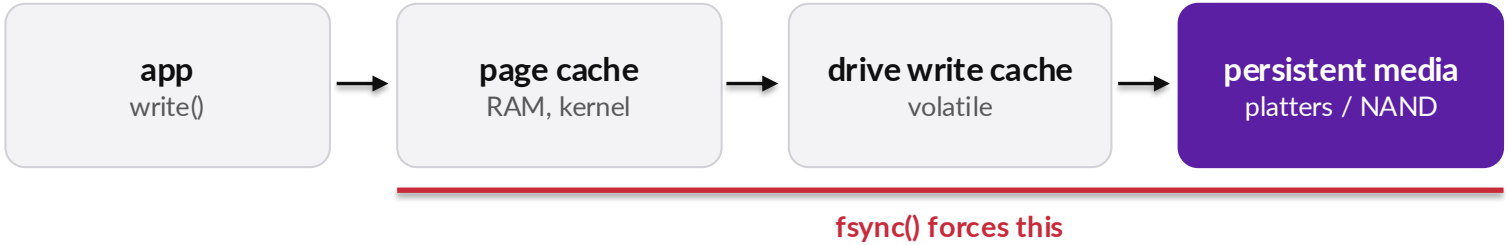
- Reads = "give me series X..Y over the last N hours"
- **Random by series**, sequential by time

Remember this slide - we'll come back to it at the end.

Write path

From write() to the platters

write() does not write to disk



- When `write()` returns, data is in RAM owned by the kernel
- Power loss means the data is **gone**
- `fsync()` is the only thing that forces data to persistent storage

fsync is expensive

Drive	fsync/s	latency
HDD 7200 RPM	40–50	20–25 ms
Samsung SSD 960 PRO	267	3.8 ms
Intel SSD DC P3700	7380	0.14 ms

Source: percona.com/blog/fsync-performance-storage-devices

Best case ~7k fsyncs/s. We need **millions of samples/s**.

How Postgres trades durability for speed

- `fsync = off`
- Unlogged tables - no WAL writes
- Bigger `max_wal_size`, `checkpoint_timeout`
- Data directory on a RAM disk

postgresql.org/docs/current/non-durability.html

What VictoriaMetrics does

- Incoming rows buffered in memory
- Every **~2s** rows become in-memory *parts* (searchable)
- Every **5s** parts are written to disk and fsync'ed
- A part = immutable, sorted, compressed - SSTable-like
- VictoriaLogs: same model, `-inmemoryDataFlushInterval=5s`

lib/storage/partition.go

```
// The interval for guaranteed flush of
// recently ingested data from memory to
// on-disk parts, so they survive process
// crash.
var dataFlushInterval = 5 * time.Second
```

lib/storage/raw_row.go

```
const pendingRowsFlushInterval =
  2 * time.Second
```

"Just use a WAL"

WAL = append every change to a log before applying it.
But does the WAL itself hit the disk?

- **Prometheus** - fsync after a big chunk of WAL is written
- **Cassandra** - `commitlog_sync_period` = 10s by default
- **Postgres** - fsync on each commit...
  but used fsync incorrectly for 20 years ("fsyncgate", 2018)

WAL: pros and cons

	WAL	No WAL (VM / VL)
Data loss window on crash	fsync period	flush interval (5s)
Write amplification	×2 (log + data)	×1
Recovery on start	replay log (minutes)	instant: open parts
Complexity	rotation, checkpoints, replay	none

It gets worse: disk caches lie

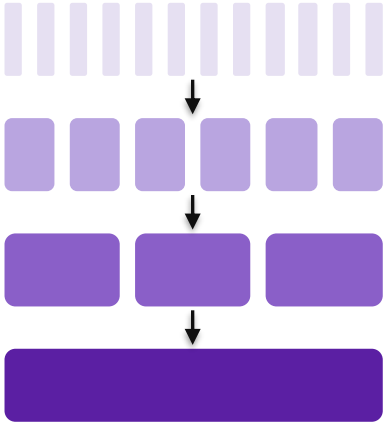
- Drives have their own **volatile write cache**
- Only enterprise drives have **power-loss protection** (capacitors)
- Some drives **acknowledge fsync without flushing**

SRE tip: check whether your cloud volume / drive has PLP before you trust "fsync = durable".

Write amplification

Merges, partitions, retention

Merges



- Fewer files, fewer open fds, faster search
- Better compression on bigger blocks
- **Cost:** each sample is rewritten several times

lib/storage/partition.go

```
// Default number of parts to merge at once.
// Obtained empirically - it gives
// the lowest possible overhead.
const defaultPartsToMerge = 15
```

Don't merge what you can't fit (VictoriaLogs)

VictoriaLogs lib/logstorage/datadb.go

```
if dstPartType != partInmemory {
  // Make sure there is enough disk space
  // for performing the merge
  partsSize := getCompressedSize(pws)
  if tryReserveDiskSpace(ddb.path, partsSize) {
    defer releaseDiskSpace(partsSize)
  } else {
    if !isFinal {
      // There is no enough disk space for
      // performing the non-final merge.
      return false
    }
    // ...
  }
}
```

- A merge needs space for **old + new** part
- At 95% disk usage a big merge can fill the disk
- Reserve space **up front**, skip the merge if it doesn't fit
- Concurrent merges share one atomic counter

Cheapest delete: drop a directory (VictoriaLogs)

`data/partitions/20261020/` - one directory per day

VictoriaLogs lib/logstorage/storage.go - dropStalePartitions()

```
n := sort.Search(len(s.partitions), func(i int) bool {
  return s.partitions[i].day >= minAllowedDay
})
ptwsToDelete := s.partitions[:n]
// ...
ptw.mustDrop.Store(true)
```

- Retention never rewrites data: no tombstones, no compaction
- Also by size: `-retention.maxDiskSpaceUsageBytes` / max disk usage %: the oldest days are dropped

Surviving power outages

Immutability, atomic rename, markers

What can be corrupted

During a merge we:

- **Write a new part** - not atomic
- **Update parts.json** - not atomic
- **Delete old part directories** - not atomic

Kill the power at any point. What does the next start see?

Rule #1: parts are immutable

- Existing part files are **never** modified
- Only **created** and **deleted**
- Worst case: a half-written *new* part that nobody references yet

Rule #2: write metadata atomically

lib/fs/fs.go

```go
func MustWriteAtomic(path string, data []byte, canOverwrite bool) {
  // ...
  // Write data to a temporary file.
  n := tmpFileNum.Add(1)
  tmpPath := fmt.Sprintf("%s.tmp.%d", path, n)
  MustWriteSync(tmpPath, data)

  // Atomically move the temporary file from tmpPath to path.
  if err := os.Rename(tmpPath, path); err != nil {
    logger.Panicf("FATAL: ...")
  }

  // Sync the containing directory, so the file
  // is guaranteed to appear in the directory.
  absPath, err := filepath.Abs(path)
  // ...
  MustSyncPath(filepath.Dir(absPath))
}
```

1. write + fsync a temp file
2. `rename()` - atomic on POSIX
3. fsync the **parent directory**

- Step 3 is the one people forget
- VictoriaLogs writes `parts.json` exactly this way

Rule #3: deleting a directory is not atomic

lib/fs/dir_remover.go

```
const deleteDirFilename = ".delete-this-dir"

func MustRemoveDir(dirPath string) {
  // Doesn't depend on atomic renaming of directories,
  // since it isn't supported by NFS and object storage.

  // Create a file indicating dirPath must be removed.
  f, err := os.Create(filepath.Join(dirPath, deleteDirFilename))
  // ...
  MustSyncPath(dirPath) // make the marker visible

  if tryRemoveDir(dirPath) {
    return
  }
  // ... schedule background retries (NFS)
}
```

Old way, removed in 2025:

rename the dir to `<dir>.must-remove`,
then delete it.
Not atomic on NFS and object storage.

Now, a marker file:

1. create + fsync `.delete-this-dir`
2. delete contents
3. delete the marker
4. delete the directory

On startup: clean up

lib/fs/dir_remover.go

```
func IsPartiallyRemovedDir(dirPath string) bool {
  des := MustReadDir(dirPath)
  if len(des) == 0 {
    // Delete empty dirs too ...
    return true
  }
  for _, de := range des {
    // ...
    if de.Name() == deleteDirFilename {
      return true
    }
  }
  return false
}
```

Plus: parts not listed in `parts.json` are removed - VictoriaLogs logs
"removed unused directory ... not listed in parts.json, which may have been left after an unclean shutdown"

Guarantees after power loss

- Existing parts are never modified, so **no corruption on merge**
- `parts.json` is updated atomically, so **the set of parts is consistent**
- Partially deleted dirs are found by the marker and **finished on restart**
- Lose at most **the last flush interval** of data

NFS

When the filesystem is a network

NFS is not a local disk

- No atomic directory rename
- **"Silly rename"**: deleting an open file leaves a `.nfsXXXX` file behind
- `rmdir` returns `ENOTEMPTY` / `EEXIST` while silly-renamed files linger
- Some servers return `EEXIST` instead of `ENOTEMPTY` (#6398)

Retry in background

lib/fs/dir_remover.go – inside MustRemoveDir

```
// NFS may perform "silly rename" before deletion,
// if client detects more than 1 file reference.
dirRemoverWG.Go(func() {
  defer func() { <-removeDirConcurrencyCh }()
  for {
    if tryRemoveDir(dirPath) {
      return
    }
    time.Sleep(time.Second)
  }
})

func isTemporaryNFSError(err error) bool {
  // Some NFS implementations return EEXIST
  // instead of ENOTEMPTY
  // ...
}
```

Watch:

- `vm_nfs_dir_remove_failed_attempts_total`
- `vm_nfs_pending_dirs_to_remove`

SRE tip:

Never mount with `nolock` or `local_lock=all` - on a stale file handle we panic on purpose

Deletes run in up to 32 goroutines: NFS latency per op is high

Instant snapshots

Immutability pays off

Immutability makes hard links possible

Parts never change. So a backup = **a set of hard links**.

- Created **instantly** regardless of size
- **Zero extra space** until the original is merged away
- Deleting the original keeps data alive in the snapshot

lib/fs/fs.go

```
func MustHardLinkFiles(srcDir, dstDir string) {
  mustMkdir(dstDir)
  for _, de := range MustReadDir(srcDir) {
    // ...
    if err := os.Link(srcPath, dstPath); err != nil {
      logger.Panicf("FATAL: ...: %s", err)
    }
  }
  MustSyncPath(dstDir)
}
```

VictoriaLogs: per-partition snapshots

VictoriaLogs lib/logstorage/datadb.go

```
func (ddb *datadb) mustCreateSnapshotAt(dstDir string) {
  fs.MustMkdirFailIfExist(dstDir)

  // flush in-memory parts before making a snapshot
  ddb.mustFlushInmemoryPartsToFiles(true)

  ddb.partsLock.Lock()
  // ... collect small + big parts, incRef()
  ddb.partsLock.Unlock()

  // Write parts.json file
  mustWritePartNames(dstDir, getPartNames(pws))

  // Make hardlinks for pws at dstDir
  for _, pw := range pws {
    fs.MustHardLinkFiles(pw.p.path, ...)
  }
  // ... decRef()
  fs.MustSyncPath(dstDir)
}
```

API

- Base: `/internal/partition/`
- `snapshot/create ?partition_prefix=2026102`
- `snapshot/list`, `snapshot/delete`
- Stale snapshots auto-removed by max age

`incRef()`: a concurrent merge can't delete parts while we link them

From snapshot to backup



- Only **new parts** are uploaded: names are unique, content is immutable
- Immutability pays off three times:
  crash safety · instant snapshots · cheap incremental backups

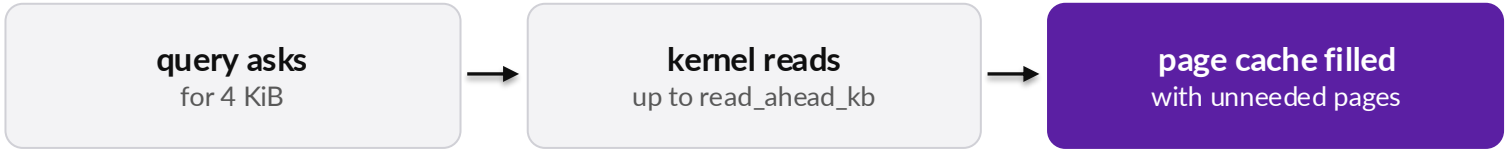
Read path
& the new feature

FADV_RANDOM / MADV_RANDOM · v1.153.0

Back to reads

- Data part files are **mmapped**
  cold reads fall back to **pread**
- A query reads many small blocks at **random offsets**
  one block per series per part

Readahead



- On a page fault / read, Linux speculatively reads **more** than requested
- Upper bound: `/sys/block/<dev>/queue/read_ahead_kb` - often 128 KiB; some cloud images & RAID: 2–4 MiB+
- Great for sequential reads, **wasteful for random ones** - issue #11461

Tell the kernel how you read

lib/fs/reader_at.go

```
func (mr *mmapReader) mustAdviseRandomRead() {
  if *disableAdviseRandomRead {
    return
  }
  if err := fadviseRandomRead(mr.f); err != nil {
    logger.Panicf("FATAL: ...")
  }
  if len(mr.mmapData) > 0 {
    err := madviseRandomRead(mr.mmapData)
    if err != nil {
      logger.Panicf("FATAL: ...")
    }
  }
}
```

lib/fs/fadvise_unix.go

```
func fadviseRandomRead(f *os.File) error {
  fd := int(f.Fd())
  err := unix.Fadvise(fd, 0, 0,
    unix.FADV_RANDOM)
  // ...
}
```

lib/fs/madvise_unix.go

```
func madviseRandomRead(data []byte) error {
  return unix.Madvise(data,
    unix.MADV_RANDOM)
}
```

Reads via `pread` need **fadvise**, `mmap` page faults need **madvise**

Only where it helps

lib/storage/part.go

```
pro.Add(timestampsPath, &timestampsFile, &timestampsSize, true)  // random
pro.Add(valuesPath, &valuesFile, &valuesSize, true)              // random
pro.Add(indexPath, &indexFile, &indexSize, false)                // keep readahead
```

lib/mergeset/part.go

```
// Keep OS readahead enabled for mergeset files.
// Most of the time searches in these files read neighboring blocks.
```

Only `timestamps` and `values` files get **RANDOM**. Index files keep readahead.

Benchmark: TSBS, cold cache, read_ahead_kb=2048

Query	Disk reads	Peak memory	p95 latency
cpu-max-all-8 (narrow: 8 hosts)	**−35%**	**−13%**	**−37%**
double-groupby-5 (broad: all hosts)	−4.7%	≈	+70–98%

- **Narrow queries** - dashboards, alerting - win big
- **Broad scans** read almost sequentially and lose readahead, so they get slower
- That's why it's **opt-in** for now

Try it

```
-fs.disableAdviseRandomRead=false    # vmsingle / vmstorage, since v1.153.0
```

Enable if	selective queries, high read_ahead_kb, disk throughput-limited, page cache pressure
Keep off if	heavy full scans, e.g. queries by metric name without label filters
Compare	disk read bytes (node_exporter), vm_fs_read_bytes_total, query p95, page cache hit ratio

Feedback: **github.com/VictoriaMetrics/VictoriaMetrics/issues/11461**
If feedback is positive, it becomes the default.

Summary

- `write()` ≠ durable. `fsync` is the price - decide consciously how much you can lose
- WAL with delayed fsync = double writes, same loss window
- Immutable files + atomic rename + fsync the directory = crash safety without recovery
- Directory delete isn't atomic - use a marker; NFS needs retries
- Immutability gives instant snapshots and incremental backups for free
- Tell the kernel how you read: `fadvise` / `madvise`

Links

VictoriaMetrics
lib/fs, lib/storage
github.com/victoriametrics/VictoriaMetrics

VictoriaLogs
lib/logstorage
github.com/victoriametrics/VictoriaLogs

Issue #11461
readahead hints
github.com/VictoriaMetrics/VictoriaMetrics/issues/11461

Issue #3038
dir removal on NFS
github.com/VictoriaMetrics/VictoriaMetrics/issues/3038

VictoriaMetrics Thank you for listening

Any Questions?